Variable data consistency in Parse Server

André Tavares
andre.s.tavares@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal
October 2017

Abstract
Modern applications resort to specialized backend storage systems to store their persistent state, such as user accounts, shared content, documents and purchases. Replication, not only across various server replicas but also on the mobile device itself, is widely adopted by these backends in order to improve the reliability and performance. Although maintaining the strong consistency among replicas can guarantee the correctness of application behaviors, it will affect the application performance at the same time because there is a well-known trade-off between consistency and performance. The documentation of Parse Server, our target backend, does not clarify the level of consistency it provides. We further investigated on this subject and we came to the conclusion that the consistency guarantees of Parse Server are dependent on how some parameters, that have impact on the consistency of its storage system, MongoDB, are being set. Unfortunately, there is no information about that configuration. We also found that tuning these parameters might not guarantee that some consistency anomalies are averted. To improve on these guarantees, we propose a solution which allows the developers, of Parse apps, to choose the consistency guarantees they require, per object, and offer a new consistency model, provided by ZooKeeper, which works as a synchronization layer between Parse Server and MongoDB. In this context, ZooKeeper is considered the "Source of truth" regarding object versions. This solution enables Parse Server to provide Sequential consistency on its objects, if developers require it. We implemented and evaluated our proposed solution in order to measure how it impacts on the latency of Parse requests and on the throughput of Parse Server. The impact in certain cases is significant, but that is the price developers have to pay in order to achieve a higher level of data consistency on their apps.

Keywords: Distributed Systems, Data replication, Data Consistency, Sequential consistency, ZooKeeper, Parse Server, MongoDB

1. Introduction

Most modern applications store data and interact with other services on the internet. For that purpose, these apps require complementary server-side platforms, often called backend, to store user accounts, shared content, documents, purchases, etc. Parse Server [1] is our target backend. It is a widely known application development platform that offers a set of tools that make it easier for programmers to focus on front-end development, taking care of things like authentication, data storage, and push notifications. Currently, Parse Server is being used by hundreds of thousands of apps [2] [3].

Data replication is widely adopted by these backends (including Parse Server) in order to improve reliability and performance. This replication happens both at the service end, where multiple server replicas provide high availability and fault tolerance, and also by creating cached copies of the data at the mobile device, for improved performance and disconnected operation. Replication can cause consistency problems between multiple copies of the data. Though maintaining strong consistency of the replicated data can guarantee the correctness of application behavior, it will affect the application performance at the same time because there is a well-known trade-off between consistency and performance, explained by the need that strong consistency protocols have to synchronize replicas before issuing a reply to the clients [4].

When trying to understand where Parse Server is placed within this trade-off, we were faced with the lack of documentation on this subject. After a deeper study, we found that Parse Server offers the consistency guarantees that its storage system, MongoDB [5], offers. So, we consulted MongoDB’s documentation and we found that the consistency guarantees that MongoDB offers are dependent on how Parse Server is using MongoDB and, in particular, how Parse Server is setting some MongoDB parameters (such as write concern, read concern and read preference), which
have an impact on the consistency of its data. Knowing that, we searched for information about that usage but we could not find any. Due to this uncertainty about how MongoDB is being used, we avoided rely on the assumption that Parse Server offers any specific consistency model, other than the fact that reads return a previously written values. Later, during the development of this thesis, we found a very interesting article that made us suspect that, even if they were setting or allowing the developers to set all those consistency related parameters from MongoDB, there are no guarantees that MongoDB provides strong consistency on its documents, particularly during server crashes and asynchrony. The article is from the website JEPSEN [6], which is a quite active and widely accepted website run by Kyle Kingsbury. JEPSEN tests clusters of datastore-type systems by firing data at them, introducing internal partitions and failures, and comparing what the system told the clients that it had stored, with what it had actually persisted, pushing vendors to make accurate claims and test their software rigorously. In this article [7] from their website, they tested different open source releases of MongoDB and report whether certain anomalies like lost updates, dirty reads or stale reads are averted in the current release. Looking at the end of this article, it is clear that the last version of MongoDB seems to avert these consistency anomalies only when the parameters related to consistency are configured in a specific way. While this is a possible route worth exploring, it was not available at the beginning of this thesis and therefore we chose a different path.

Since we cannot expect any consistency model from either Parse Server or the underlying storage systems that it uses, we aim to improve Parse Server consistency guarantees by extending it to support a more principled approach towards providing consistent access to the data it stores. Specifically, we want to build on recent proposals for multi-level consistency [8], which allow the developer (or an automated tool on their behalf [9]) to associate an appropriate consistency level to each Parse object. Thus, we think that exposing consistency to developers this way could be a very interesting design decision because it allows them to be more aware of consistency issues, gives them more control, and allows for consistency to become a first order concern in the programming of applications.

2. Related Work
We next present and discuss our research on previous work related to providing different levels of consistency, in order to have a good insight about the possibilities and existing solutions for the problem we are trying to solve.

2.1. Simba: Tunable End-to-End Data Consistency for Mobile Apps
Simba [10] presents sTable, a high-level programming abstraction, design for developing mobile applications connected to the cloud. It provides tunable end-to-end data consistency, with a unified data model for table and object data, and atomicity over coarse-grained and inter-dependent data, meaning data granularity relevant in the application. For example, data related to an email. To demonstrate the utility and practicality of sTables, they created Simba, a data synchronization service. Simba treats a table as the unit of consistency specification and a row as the unit of atomicity preservation. Each sTable has associated one of three consistency levels: Sequential, Causal and Eventual. Programmers define the consistency level per table, for example, a table of images, where each row can be viewed as an application-level object, composed by the collection of objects related to a specific image, and each row is treated atomically in every read and write.

Despite the fact that sTables, is a well define programming abstraction with flexibility concerning data consistency, adapting existing Parse Server applications to use sTables can be quite challenging because programmers have to change how data is organized and accessed, which will potentially induce considerable changes on existing applications.

2.2. Putting Consistency Back into Eventual Consistency (Explicit Consistency)
The authors present Explicit Consistency [11], a consistency model that ensures application correctness, centered on its semantics, and not on the order of operations. It offers eventual consistency by requiring application-specific invariants, which are defined by the programmers. Explicit Consistency identifies which operations could not be safe to execute concurrently, and allows the programmers to choose between violation-avoidance or invariant-repair techniques. Violation-avoidance is achieved by relying on a reservation system that moves replica coordination away from operation execution. Invariant-repair, in turn, allows operations to execute without restriction, and restores invariants by applying a repair operation to the database state. The system is free to reorder execution of operations at different replicas, provided that the specified invariants are maintained.

Explicit consistency requires programmers to make some considerable changes (define invariants, violation-avoidance and invariant-repair techniques) on existing applications, which we want to avoid at most. Another disadvantage is that the
programmer does not have any control over the consistency guarantees of each operation or object.

2.3. Customizable and Extensible Deployment for Mobile/Cloud Applications (Sapphire)

Sapphire [12] is a distributed programming platform that simplifies the programming of mobile and cloud applications by separating the application logic from the deployment logic. Sapphire’s key design feature is its distributed runtime system, which supports a flexible and extensible deployment layer for solving complex distributed tasks, such as data consistency, fault-tolerance, code-offloading and caching. Rather than writing distributed systems code, programmers choose from a list of deployment managers that extend Sapphire’s kernel to meet their applications’ deployment requirements. They can also create custom deployment managers. With this flexibility, programmers can quickly switch deployment solutions to respond to environment or requirement changes, or simply to test and compare alternatives during development.

Sapphire, in terms of the interface for mobile applications, takes a clean slate approach and tries to develop an API that exposes, to the application developer, several aspects such as consistency, caching, replication and scalability. However, we are more interested in an approach that privileges the seamless adaptation of existing code, by avoiding changes on the Parse Server API as much as possible.

2.4. HyperDex: A Distributed, Searchable Key-Value Store

HyperDex [13] is a high-performance, scalable, consistent and distributed key-value store that combines strong consistency guarantees with high availability in the presence of failures and partitions affecting up to a certain number of servers. HyperDex uses a replication protocol called value-dependent chaining, to simultaneously achieve fault tolerance, high performance and strong consistency. Regarding the consistency of the keys, HyperDex guarantees that all operations on a specific key (e.g., get and put) are linearizable [14] with all operations on all keys. Concerning consistency when searching, HyperDex guarantees that a search will return all objects that were committed at the time of search, meaning that an application whose put succeeds is guaranteed to see the object in a future search. When concurrent updates are occurring, a search may return either the committed version or the newly updated version of an object which matches with the search query.

Replacing MongoDB with HyperDex could be a solution. HyperDex is a completely different database from MongoDB, but it a has a compatible API (i.e. it can act as a stand-in for MongoDB), provides strong consistency in all configurations and the authors say that it is 1-4x faster than MongoDB itself [15]. However, this API does not support some MongoDB features, and there is no available information about which ones and so we can not argue if they are important ones or not. Another important aspect to consider is that HyperDex has a much smaller user base, is much more specialized and it seems both development on the product and commercial support has stopped. In contrast, MongoDB is one of the most popular databases and it is used by thousands of companies so it has the advantage of being maintained, available, supported and it has a very large community around it.

2.5. ZooKeeper: Wait-free coordination for Internet-scale systems

ZooKeeper [16] is a replicated synchronization service for coordinating processes of distributed applications. It is used for maintaining information about configuration, naming, provide distributed synchronization and group services. The interface exposed by ZooKeeper has wait-free data objects, organized hierarchically as in file systems, with an event-driven mechanism, similar to cache invalidations of distributed file systems, in order to provide a simple, yet powerful coordination service. It is robust, since, the persisted data is distributed between multiple nodes (this set of nodes is called an “ensemble”) and one client connects to any of them (i.e., a specific “server”). As long as a strict majority of nodes are working, the ensemble of ZooKeeper nodes is alive. Thus, consensus, group management, and other related protocols are implemented by this service so that the applications do not need to implement them on their own.

Regarding its data consistency guarantees, ZooKeeper provides a guarantee of FIFO execution of requests per client, and linearizability for all requests that write on ZooKeeper. Each time a client writes to the ensemble, a majority of nodes persist the information. These nodes include the server with which the client is connected, and also the leader, which is the server that contains the full state of Zookeeper. This means that each write makes the server up-to-date with the leader. To guarantee that write operations satisfy linearizability, ZooKeeper uses a leader-based atomic broadcast protocol [17]. Reads are concurrent since they are served by the specific server that the client is connected. However, the "view" of a client may be outdated, since the leader updates the corresponding server with a bounded but undefined delay. Both read and write operations are designed to be fast, though reads are faster than writes. In the following list are the consistency guarantees pro-
vided by ZooKeeper:

- **Sequential Consistency**: Writes from a client will be applied, in all replicas, by the order that they were sent.

- **Atomicity**: writes either succeed or fail, meaning that there are no partial writes.

- **Single System Image**: A client will see the same state of ZooKeeper regardless of the server that it is connected.

- **Reliability**: Once a write to a node has been committed, it will persist until a client overwrites that node.

- **Timeliness**: The clients view of the system state is guaranteed to be up-to-date within a bounded interval on the order of tens of seconds.

Implementing a synchronization layer between Parse Server and MongoDB is also a possible solution. Instead of relying on MongoDB consistency, Parse Server can use an external coordination system, such as ZooKeeper, that offers Sequential consistency guarantees, to validate the consistency of MongoDB objects, by consulting and comparing "version values", that will be stored on both systems. Despite the fact that this extra "layer" can potential induce some delay on Parse requests, the impact on performance is expected to not be significant since ZooKeeper has great performance on writes and specially on reads, and also because the files that would be stored in it would be very small.

### 3. Solution

Our decision to strengthen the consistency guarantees of Parse Server was to use an external coordination service, ZooKeeper, to provide another consistency model to Parse Server. The developers of Parse apps can now decide if they want Parse Server to provide a clear consistency model on their objects, Sequential consistency. This choice is justified by the various advantages we presented in the section 2.5, which outweigh the disadvantages of the other alternatives. Specifically, because of the Sequential guarantees ZooKeeper provides, because of the ease to use it in the context of this problem, and because of its performance.

In terms of the interface that our new system offers, we decided to expose consistency to developers by enabling Parse Server to provide two consistency levels: the Default one, provided by MongoDB, and Sequential consistency, which corresponds to the level of consistency that ZooKeeper is configured to provide on each deployment. For the first consistency model, we call it "Default" because it is not possible to give a more precise name, since there is no information about the consistency guarantees provided by Parse Server.

The idea behind using ZooKeeper is the following: On each read and write, our code gets and compares version numbers, associated to the MongoDB objects, that are stored on both MongoDB and ZooKeeper. Thus, in the context of this problem, Zookeeper is considered the "Source of truth", meaning that in every read and write on MongoDB objects, those operations are first validated by querying the most recent version of the target object, that is stored on ZooKeeper, in order to ensure Sequential consistency on the target object. To implement this extra consistency model, we have to make some changes on Parse Server to also call ZooKeeper and compare those object versions. This changes are minimal and, above all, the majority of the changes is confined to a single location in the source code. Specifically, we need to change the file that make calls to MongoDB.

Regarding the changes on the code from the client side, meaning on the SDKs code, we decided to only change on Parse SDK since that is sufficient to evaluate our proposed solution. We made minor changes on the Parse Android SDK in order for the programmer specify the consistency model for each Parse object.

#### 3.1. Architecture

In this section we present the architecture of our proposed solution, which gives an overview of the main blocks of the whole system. This high-level architecture is depicted in Figure 1. The key change to the original architecture of Parse Server is that now, Parse Server instances are not only connected to a MongoDB replica set but also to a ZooKeeper ensemble. The following list presents and describes the main components of the system:

- **Clients** interact with applications that are using the modified Parse SDKs to make requests to Parse Server instances

- **Load balancer** distributes the workloads across all Parse Server instances

- **Parse Server instances** are back-end servers that serve the clients

- **MongoDB replica set** is the storage system where Parse Server stores all users data

- **ZooKeeper ensemble** is a replicated synchronization service that keeps record of the most recent version of each object stored on MongoDB
3.2. Design

In the context of our problem, ZooKeeper, as "source of truth", determines what is the most recent version of each object stored on MongoDB. Considering that, the main design idea is the following: If the programmer does not set the consistency for the target object or if it sets it to "Default", then the Parse Server will behave has it was behaving before, that is running the original calls to MongoDB and no calls to ZooKeeper neither run additional code; If the programmer specifies that he/she wants Sequential consistency for the target object, then Parse Server will make calls to ZooKeeper and run additional code, which can be compare versions, generate hashes, etc, depending on the type of request. In our proposed design, a version of an object is represented by an hash of it. Instead of an hash, we could also use just a random number, but we decided to use an hash. We present next our main design idea, by describing the control flow of actions for each type of Parse request, when Sequential consistency is required:

Create Object (POST) :
1. Store the hash on ZooKeeper;
2. Store the object + the hash on MongoDB;

Get Object (GET) :
1. Get the object from MongoDB;
2. Get the hash from ZooKeeper;
3. If the hashes match, then return the object;
4. Else, use the hash from ZooKeeper to get the correspondent object from MongoDB;

Update Object (PUT) :
1. Get the object as previously explained;
2. Store a copy of it, using its hash as id;
3. Update the hash on ZooKeeper with a new one;
4. Update the object on MongoDB as normal, but also update the hash on it with a new one;

Delete Object (DELETE) :
1. Get the object from MongoDB;
2. For each old version value in it, delete the correspondent object from MongoDB;
3. Delete the hash from ZooKeeper;
4. Delete the object from MongoDB;

4. Evaluation

In the following experiments, we introduce the concept of "inconsistency". Our solution interprets the concept of "inconsistency" as when, on a GET or PUT request, the hash retrieved from ZooKeeper, which represents the most recent version of its correspondent object, it is not equal to the hash stored on the correspondent object, retrieved from MongoDB.

We evaluated our proposed solution in order to measure how it impacts the latency of Parse requests and on the throughput of Parse Server. About the results without inconsistencies occurring, overall they show an impact on latency and throughput that is not significant, when requiring Default vs Sequential consistency. However, the results on the PUT request experiments show a significant impact on the average latency and throughput, as depicted on Figures 2, 3 and 5. Specifically, there exists a decrease of around 76% on the average throughput and an increase between 50 to 74 percent on the average latency. Regarding the results when inconsistencies occur, at a first glance, they seem to show a very significant impact on latency. When requiring Default vs Sequential consistency, there exists an increase of around 98% on the average latency of GET requests, and an increase of around 114% on the average latency of PUT requests. However, when carefully analyzing these results, one needs to take into consideration that there is no knowledge about what is the percentage of inconsistencies in the total amount of requests to Parse, because that number depends on many variables, and so we can not truly know what is the real impact of inconsistencies on the overall latency and throughput of Parse Server since it depends on how each Parse application is using it. Thus, we came to the conclusion that the impact of inconsistencies can only be calculated per application, by measuring the latency and throughput on real world deployments,
with real users, or something that simulates that real world behavior.

![Figure 2: Default consistency vs Sequential consistency](image1)

![Figure 3: Default vs Sequential w/o inconsistencies vs Sequential w/ inconsistencies](image2)

5. Conclusions

We think that our proposed solution is a viable extension to improve the data consistency of Parse Server since: it offers an alternative consistency model to the developers, besides the one that Parse Server already offers; That consistency model is provided by ZooKeeper, which yield strong guarantees on the consistency of its data; The modifications to Parse and its SDKs are minor and local; and the impact on the latency and throughput of Parse Server is the price that developers have to pay in order to achieve a higher level of data consistency on their apps.

References


